An Essay on Real-Time Databases

Raul Barbosa
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
rbarbosa@ce.chalmers.se

1 Introduction

Data management in real-time systems has traditionally been implemented via application-dependent designs. The main drawback of this approach is that, as the applications grow in complexity and amount of data, the code which deals with data management becomes increasingly difficult to develop and maintain. Real-time database systems are the most promising alternative to manage the data with a structured and systematic approach.

Database systems have been successfully applied in many production systems. Many websites rely on conventional databases to provide a repository for data and to organize the retrieval and manipulation of data. However, this type of database is not suitable for applications with timing requirements (e.g. air traffic control, telecommunications and industrial control). It thus becomes interesting to apply techniques from the real-time systems research to provide timeliness in addition to the common features of database systems.

The typical database design aims to provide a fast response time. Indeed, with suitable hardware resources it is possible to achieve impressive results. IBM presented in late 2006 a system capable of completing more than 4 million transactions per minute when running the TPC-C benchmark [15]. The same system would be even faster if advanced performance optimization techniques (such as indices) were allowed by the TPC-C specification.

A fast database is, however, not necessarily a real-time database. For some applications the average transaction rate is not as important as completing each transaction before a certain deadline. For such applications the transaction rate is a parameter specified by the application designers. The real-time database then has to be designed in such a way that transactions meet their deadlines.

Real-time databases thus have the requirement of ensuring transaction timeliness in addition to the well-known ACID properties. According to the ACID model the key transaction processing properties are *atomicity*, *consistency*, *isolation* and *durability*. A database management system which does not ensure all these properties (without a solid argumentation) should not be considered trustworthy.

- **Atomicity** – A transaction either completes all its operations or none of them. If any operation fails then the transaction must be rolled back.
- **Consistency** – Any transaction which breaks the integrity constraints of the database must be aborted and its actions rolled back.
- **Isolation** – Transactions which are executed simultaneously must not influence each other. All transactions "see" the database as it was before any other transaction started or after it committed.
- **Durability** – The result of any committed transaction will be persistent in the database, even in the event of a failure.

The ideal real-time database should be able to perform real-time ACID transactions. However, the timeliness requirement poses an additional challenge which often implies relaxing one or more ACID properties in order to complete the transactions in a timely manner. It is therefore necessary to consider the *timeliness* aspect of the transaction processing mechanisms at the same level as the ACID properties. A real-time database should therefore be evaluated based on the support for TACID transactions [16].

2 Design Issues

Predictable response times often require the database to be *memory resident*. Placing the database on main memory avoids the non-determinism of disk accesses. At the same time the performance of the database is significantly improved. On the other hand, the *durability* of data is not guaranteed in the event of a failure unless techniques such as checkpointing, journaling or replication are used. These techniques incur additional overhead during normal execution as well as system recovery and should therefore be carefully analyzed. This means that timeliness and durability are in some sense competing properties. When the hardware resources are scarce the durability is often relaxed in order to ensure timeliness.

Scheduling transactions in order to satisfy their deadlines is where the real-time research makes a fundamental contribution. Real-time databases make use of well established scheduling policies such as the *Rate Monotonic* (RM) algorithm and the *Earliest Deadline First* (EDF) algorithm, both presented in [11]. With the rate monotonic algorithm the transactions have a fixed priority inversely proportional to their periods. The rate monotonic policy is an optimal fixed priority scheduling algorithm. This means that if a given set of transactions can be scheduled with a fixed priority scheme, then it will be schedulable with the rate monotonic algorithm. The EDF algorithm is an optimal dynamic priority scheduling algorithm which assigns the highest priority to the transaction which has the closest deadline. The RM algorithm is suitable for a system with periodic transaction whereas the EDF approach suits both periodic and non-periodic transactions.

The issues involved in the design of real-time databases sometimes deal with other areas of research. The usual real-time system architectures do not place the database on a dedicated computer system. This extends the problem of determinism to the *operating systems* which manage the computer system's resources. In order to ensure transaction timeliness one must use real-time operating systems. Only this approach can ensure that the tasks which serve the transactional mechanisms are executed in real-time.

Similarly, if the database has a distributed nature then the message passing mechanisms must provide real-time guarantees. Network standards such as Ethernet are not suitable for distributed real-time database systems. In a general sense all the layers underlying the database system must offer real-time services in order to ensure that the database has real-time properties.

It is arguable that the technological advances in hardware will solve the problem of designing real-time databases [14]. There is certainly a benefit to be taken from hardware improvements. Solid state drives may be non-volatile (e.g. based on Flash technology) and provide very fast and deterministic data accesses. Using this type of data storage would ensure the durability of data without compromising the timeliness of the transactional mechanisms. Parallel processors will also help in obtaining more powerful computing platforms.

Nonetheless, hardware alone does not guarantee that a conventional database will provide real-time response. It is, for instance, necessary to enhance the existing database interfaces - such as the Structured Query Language (SQL) - with support for the specification of timeliness requirements along with the queries.

Furthermore, even if the database system solves all real-time issues, this does not ensure that the query itself is executable within the specified deadline. The database design process requires the help of Worst Case Execution Time (WCET) analysis tools to determine how long will it take for a transaction to complete. This is an extremely complex problem as it may be difficult or overly pessimistic to predict the execution time under concurrency control conditions. In addition to this, the algorithms used to obtain data from the database may, for instance, involve full table scans (e.g. when searching for a specific row without using an index). This implies that the execution time depends on the number of rows present in the table at any given time. Consequently, static analysis methods will not help in determining the WCET of a transaction, unless the maximum number of rows in the table is known *a priori.*

3 Concurrency Control

Due to the nature of the applications, database systems have to deal with concurrent transactions. Resolving data conflicts among transactions is an issue for *real-time concurrency control* mechanisms. The importance of this issue motivates a thorough discussion.

It has been suggested that, in uniprocessor systems, one can simply execute the transactions sequentially. Thus, concurrency control would not be necessary. The main advantage of this approach is that the overhead of concurrency control mechanisms would be eliminated. Furthermore, the overall number of context switches due to blocking would be decreased. However, concurrency is required when transactions vary both in length and priority. If a high-priority transaction is issued when a long running low-priority transaction is already executing, it is only possible to ensure timeliness if concurrency is enabled.

The main requirement is that *serialization consistency* must be guaranteed. This means that the result of the read/write operations must be the same as if the transactions had been executed in serial order. It should be noted that there is no requirement on the actual order, as long as it is possible to ensure *some* ordering. Ref. [9] discusses this issue in detail.

There are two main classes of concurrency control techniques: *optimistic* and *pessimistic.* In the optimistic approach the operations are carried out and the transaction is checked for serialization consistency before the results may be written into the database. The pessimistic approach allows the transaction to start executing only when it is ensured that serializability will not be violated.

Two-phase locking – a pessimistic concurrency control technique – is the most common approach for concurrency control in centralized databases. The procedure involves the use of locks in order to gain the access to data items. In the first phase the transaction must acquire the locks on the required data items. For optimization purposes these locks can be for reading only as well as for writing. In the second phase the transaction releases the locks. With two-phase locking a lower priority transaction T_L is able to block a higher priority transaction T_H when T_L has the lock on a data item which T_H needs and T_L is allowed to complete before T_H can access it. This can cause *priority inversion.*

The problem of priority inversion arises when transaction T_H is waiting for T_L to release the lock on data item I and a transaction T_M with an intermediate priority arrives to the system. If T_M does not require access to data item I then it will preempt T_L, thus delaying T_H. Their priorities are therefore inverted.

The simplest solution to the problem is based on the fact that transactions can usually be aborted and restarted. The *high priority protocol* [1] aborts low priority transactions when higher priority transactions require data items which are locked. With this protocol a transaction is never blocked by a lower priority transaction – the protocol is deadlock-free. On the other hand, any work done by an aborted transaction is wasted and needs to be re-executed.

Another solution to the priority inversion problem is to make a low priority transaction inherit the priority of the higher priority transaction when there is a data conflict between them. This is known as the *priority inheritance protocol* [12]. In these conditions the low priority transaction is said to be executing on behalf of the higher priority transaction. The protocol prevents priority inversion since only a transaction with a priority higher than both transactions can preempt the low priority transaction (and delay the high priority transaction). However, the drawbacks of priority inheritance are twofold: i) the highest priority transaction may, in the worst case, be blocked once by every other transaction and ii) deadlocks are not prevented (e.g. if locks are acquired in reverse order).

When a deadlock occurs one must abort and restart one or more transactions in order to ensure that their execution progresses and eventually comes to an end. There are many different criteria for choosing which transactions should be aborted. However, it has been verified empirically that, in most cases, deadlocks involve only two transactions [4]. Thus, any such criteria should be inexpensive to implement as there not many ways to choose the wrong victim.

The *priority ceiling protocol* [2] overcomes both drawbacks of priority inheritance. The *priority ceiling* of a data item is defined as the highest priority among the transactions that may lock the data item. For a transaction to execute, it must have a priority *strictly higher* than the highest priority ceiling of all data items locked by all other transactions. The priority ceiling protocol overcomes the two shortcomings of the priority inheritance method – deadlocks are prevented and high priority transactions are blocked by at most one transaction.

Optimistic concurrency control techniques [5, 7] divide the execution of the transactions into three phases. The first is the *read phase*, where the required data items are read. The second is the *validation phase*, detailed in [8], where the system must check if serialization consistency may be violated if the data is written into the database. Finally, in the *write phase* the data is written into to the database. This class of protocols is considered very good for real-time databases as it is deadlock-free and non-blocking. The main shortcoming is that one or more transactions may need to be aborted as the consequence of a failed validation (and any completed work needs to be restarted).

Optimistic and pessimistic concurrency control techniques provide the means to guarantee the serialization consistency of transactions. However, serialization consistency can be sacrificed in favor of timeliness [13]. With this approach the transactions access the data items concurrently *without control*. This idea is only applicable to systems where the data is refreshed at a high enough frequency. The rationale behind this strategy is that any inconsistent state will not be propagated throughout the database or the amount of error will be tolerable. This is another example where the ACID properties of transactions are relaxed in order to ensure their timeliness.

4 How Hard is Real-Time?

Transactions have many attributes (priority, execution time, resource usage, etc.), the most important of which is the *deadline*. A transaction's deadline is the mean by which the system designer can specify its timeliness requirement.

A way of classifying transactions is based on the effects of their deadlines not being met. *Hard real-time* transactions are those which can have catastrophic consequences if they are not executed on time. *Soft real-time* transactions, on the other hand, will only degrade the system's performance when not completed within their deadline. A special type of soft real-time transaction is a *firm real-time* transaction, which has no value to the system when its deadline is not met and should therefore be aborted.

Transactions may also be classified according to the periodicity of their arrival – a very important property for schedulability analysis. Some applications require certain transactions to be executed periodically, with a given frequency. *Periodic* transactions can thus be statically scheduled to execute at a fixed rate. In contrast, there are *aperiodic* transactions that must be scheduled whenever they arrive. Aperiodic transactions offer little or no predictability and they are suitable for scheduling with dynamic policies (or with static policies which have enough computing power open for competition among aperiodic transactions). A special kind of aperiodic transactions are *sporadic* transactions, which have a known minimum inter-arrival time. Schedulability analysis for sporadic transactions is somewhat easier than for pure aperiodic transactions.

It is often a requirement that a database system must deal with mixed types of transactions. In the extreme case there may be real-time transactions executing in parallel with transactions which have no timing requirements. Furthermore, periodic and non-periodic transactions may coexist. The database system must therefore support different notions of time and criticality. Designing a generic real-time database is therefore a very complex task. There is an extremely large amount of ways in which one can model transactions. Hopefully, it will be possible to devise a way which satisfies most application scenarios but, naturally, there will always exist application-specific issues which must be dealt in an application-specific manner.

5 Memory Resident Databases

Memory resident databases help in improving the performance of the system and, most importantly, do away with the non-determinism of disk accesses. Such an approach is making its way into mainstream databases designed by companies which target time-critical businesses. The applications include operational support systems, airline and reservation systems, etc.

As mentioned earlier, the durability of the stored information cannot be ensured unless techniques such as journaling, checkpointing or replication are used. Since the main memory is volatile, a power outage is enough to wipe out all data. Conventional databases rely on the fact that the integrity of the disk subsystem is usually not affected by a system crash.

Due to all the issues discussed so far (concurrency control, scheduling, technological limitations, execution time analysis, etc.) and many other, not discussed in this essay, placing the entire database in memory is not sufficient to make it suitable for real-time applications.

On the other hand, there are specialized disk scheduling algorithms which help improving the determinism of disk accesses. Thus, it is not mandatory to place the *entire* data in main memory. This is especially beneficial for applications which manage very large amounts of data. Nevertheless, performance goals will always require the *right data* to be in main memory at the *right time*. An approach commonly taken is to enlarge the buffer pool, thereby providing enough space to retain most or perhaps the entire set of data items required by the transactions.

6 Hard Real-Time Databases

A pragmatic way to design real-time databases is to minimize the ratio of transactions that miss their deadlines. This strategy is, however, a step short of meeting hard real-time requirements, under which missing a single deadline can lead to catastrophic effects. Most commercial databases that claim real-time properties are not suitable for hard real-time applications.

Real-time database designers are faced with a vast number of constraints, difficult to fulfill while offering an adequate set of features. The database source code, developed to manage the transactional mechanisms, has to be thoroughly analyzed in order to guarantee worst case execution timings. Furthermore, a generic hard real-time database would have to suit a great diverseness of application requirements, thereby demanding a significant development effort.

In addition to this, due to their criticality, hard real-time applications usually have dependability demands. One might, for instance, need to ensure a certain level of *availability*, i.e. the fraction of time during which the system is in an operational state, or a particular *mean-time-to-failure* (MTTF) for the system. To achieve these goals the dependability community has devised a number of techniques, suitable for different applications. Again, it is costly to develop a generic fault-tolerance framework to support any database application.

Providing hard real-time transactions limits the broadness of the set of features a database can offer. A particular method to *join* two tables may have an order of complexity which makes its execution depend on the number of rows in the table or require too much memory. If there is no other algorithm capable of providing the same functionality, with a feasible amount of computation, then it may be difficult to make it part of a hard real-time database.

However, many different research efforts have been presented in the past years which may lead to functional hard real-time databases. At the moment there are many commercially available databases which are time-cognizant and feature a great deal of predictability. Such databases already solve to some extent the issues debated in this essay. They should be expected to evolve with the goal of performing hard real-time ACID transactions.

7 Index Structures for Real-Time

Indices are, in the taxonomy of databases, data structures which improve significantly the performance of search operations. Indices of diverse types integrate conventional database management systems with the goal of providing fast response times. Due to this fact, indices assume a vital part of real-time databases. Some index structures provide, in addition to performance improvements, a predictable access time – which is highly desired in real-time systems.

Real-time indexation requires specialized structures which maximize the usage of CPU and main memory [10]. The most common index structure is the *B+ tree* – a variant of the B tree. B trees are broad and not very deep (a common tree does not go beyond 3 levels of depth). This makes them suitable for disk databases. However, if the data is in main memory the B tree should be preferred to the B+ tree. B+ trees keep the data in the leaves of the tree. Instead, the leaves should store the pointers to the actual data, with the purpose of saving storage space. Following a pointer in memory is far less computationally expensive than doing so on disk. Figure 1 shows the structure of a B tree node.

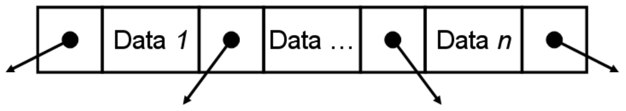

Figure 1. Structure of a B-tree node

Another structure which has had some application in the past is the *AVL tree.* AVL trees are binary search trees which are kept height-balanced through rotation operations (whenever an update breaks the balance). Since AVL trees are binary, there are two pointers (left and right) at each node. Every node stores a single data item (or a pointer to a data item). AVL trees are therefore less convenient than B trees due to the storage requirements. The structure of an AVL tree node is depicted in Figure 2.

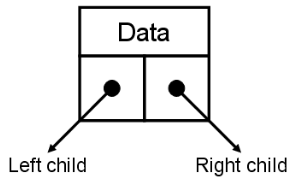

Figure 2. AVL-tree node structure

The *T tree* structure [10] is a mix of B trees and AVL trees. It is a binary tree but each node stores many elements, thereby keeping the favorable storage properties of B trees. Rebalancing the tree on update is also required but, since there is the possibility of moving data *within* each tree node, it does not happen as frequently as with AVL trees. There is, however, a disadvantage with this structure: traversing the tree is computationally expensive. Figure 3 shows the structure of the nodes of T trees.

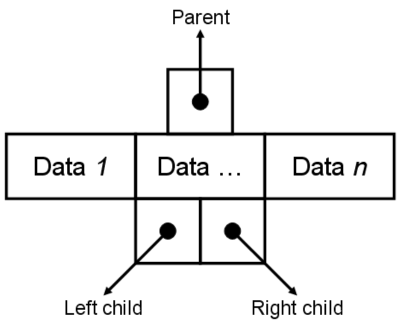

Figure 3. The structure of a T-tree node

The *T* tree* is a variation of the T tree, proposed in [3]. It uses the basic structure of the T tree, with an additional pointer to the successor node. The successor pointer is depicted in Figure 4. This pointer makes sequential search work in a similar way to *linked lists.* The performance of T trees is therefore improved in certain situations.

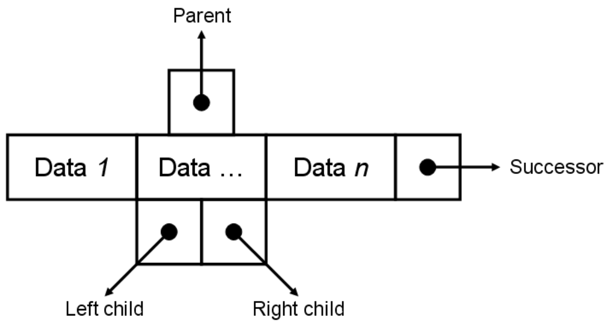

Figure 4. T*-tree node structure

7.1 Index Concurrency Control

The high throughput desired in some applications may raise the contention in accessing the indices to the point where it becomes a more serious performance bottleneck than the normal data accesses. In addition to the regular concurrency control issues, described in Section 3, we must be aware of the problems with index concurrency control.

The study presented in [6] gives some insight into the problem, by describing conventional index concurrency control mechanisms with the incorporation of priorities, in the context of firm real-time transactions. B tree concurrency control is achieved through the use of locks on index nodes (latches). Incorporating priority into this type of technique is done by preempting the lower priority transaction when a conflict is detected. This transaction will be required to repeat the last index operation (not all transaction operations). On the other hand, a lower priority transaction will never acquire a lock if there is a conflict.

There is also the possibility of using priority inheritance, which uses the same method described in Section 3. It also comes with the same pros and cons of conventional priority inheritance – there will be no wasteful restarts but there is a possibility of blocking high priority transactions when lower priority transactions already acquired the locks.

8 Conclusions

Real-time database systems are emerging as a structured and systematic approach to manage data, avoiding application-dependent designs. This approach combines techniques from the database systems community with the existing methods for real-time computing.

At least part of the database must be placed in main memory, in order to avoid the non-determinism of disks. However, a fast database executing in main memory is not necessarily a real-time database. For some applications completing each transaction within its deadline is more important than the average transaction rate. This requires a specialized design which ensures that transactions meet their deadlines.

Indices are data structures which improve significantly the performance of search operations. In some cases the order of complexity of the search operations is reduced, which means that the database will be able to provide additional features, impossible to offer otherwise. There are several different types of index – some more appropriate than others for real-time. The T* tree and the B tree indexes are the most suitable choice for this type of system.

The problem of concurrency control, in accessing data items as well as index nodes, has received some attention. Real-time performance, while ensuring the consistency of data, is a delicate problem, solved by a few distinct strategies (each with advantages and drawbacks). In some cases it is even possible to neglect concurrency control by ensuring that the data is being refreshed quickly enough and that the amount of error propagated through the database is small.

Designing a hard real-time database is not a simple task, given the vast number of constraints and the diverseness of application requirements. Real-time databases which strive to provide timeliness in addition to the well-known ACID properties are often required to sacrifice one or more such properties (e.g. timeliness vs. durability). A common strategy is to minimize the ratio of transactions that miss their deadlines. However, this may not be applicable to hard real-time systems, where missing a single deadline can have catastrophic effects.

At the moment there are many commercially available databases which are time-cognizant and feature a great deal of predictability. Most of these databases are not yet suitable for hard real-time applications. Nonetheless, the numerous research efforts currently ongoing may lead to functional hard real-time databases in the near future.

References

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, Sept. 1992.

[2] M.-I. Chen and K.-J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[3] K.-R. Choi and K.-C. Kim. T*-tree: a main memory database index structure for real time applications. In *RTCSA*, pages 81–88. IEEE Computer Society, 1996.

[4] C. Devor and C. R. Carlson. Structural locking mechanisms and their effect on database management system performance. *Information Systems*, 7(4):345–358, 1982.

[5] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the Real-Time Systems Symposium*, pages 94–103, Lake Buena Vista, Florida, USA, Dec. 1990.

[6] J. R. Haritsa and S. Seshadri. Real-time index concurrency control. In *Real-Time Database Systems*, pages 59–74, 2001.

[7] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th Conference on Very Large Databases*, pages 35–46, Sept. 1991.

[8] H. F. Korth and A. Silberschatz. *Database System Concepts*. McGraw Hill, New York, NY, 1991.

[9] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill International Editions, 1997.

[10] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Kyoto*, Aug. 1986.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 1(20):44–61, Jan. 1973.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky. Concurrency control for distributed real-time databases. *ACM SIGMOD Record*, 17(1):82–98, Mar. 1988.

[13] M. Singhal. Issues and approaches to design of real-time database systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(1):19–33, Mar. 1988.

[14] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, June 1999.

[15] Transaction Processing Performance Council. TPC benchmark C standard specification (revision 5.8.0). Dec. 2006.

[16] M. Vieira, A. Casimiro, and H. Madeira. TACID transactions. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, 1st Workshop on Hot Topics in System Dependability (HotDep-05)*, pages 293–298, Yokohama, Japan, June 2005.